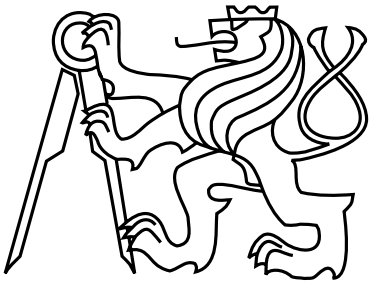
České vysoké učení technické v Praze
Fakulta stavební
Katedra geomatiky



Cvičení 2

Úloha č. 2: Konvexní obálky a jejich konstrukce

Bc. Petr Poskočil a Bc. Marek Fáber

Vyučující: doc. Ing. Tomáš Bayer, Ph.D.

Studijní program: Geodézie a kartografie, Navazující magisterský

Obor: Geomatika

6. listopadu 2019

Obsah

Kapitola 1

Zadání

Úloha č. 2: Konvexní obálky a jejich konstrukce

Vstup: množina $P = \{p_1, ..., p_n\}$, $p_i = [x, y_i]$.

Výstup: $\mathcal{H}(P)$.

Nad množinou P implementujete následující algoritmy pro konstrukci $\mathcal{H}(P)$:

- Jarvis Scan,
- Quick Hull,
- Swep Line.

Vstupní množiny bodů včetně vygenerovaných konvexních obálek vhodně vizualizujte. Pro množiny $n \in < 1000, 1000000 >$ vytvořte grafy ilustrující doby běhu algoritmů pro zvolená n. Měření proveďte pro různé typy vstupních množin (náhodná množina, rastr, body na kružnici) opakovaně (10x) a různá n (nejméně 10 množin) s uvedením rozptylu. Naměřené údaje uspořádejte do přehledných tabulek.

Zamyslete se nad problematikou možných singularit pro různé typy vstupních množin a možnými optimalizacemi. Zhodnoťte dosažené výsledky. Rozhodněte, která z těchto metod je s ohledem na časovou složitost a typ vstupní množiny P nejvhodnější.

Hodnocení:

Krok	**Hodnocení**
Konstrukce konvexních obálek metodami Jarvis Scan, Quick Hull, Seep Line.	15b
Konstrukce konvexní obálky metodou Graham Scan	*+5b*
Konstrukce striktně konvexních obálek pro všechny uvedené algoritmy.	*+5b*
Ošetření singulárního případu u Jarvis Scan: existence kolineárních bodů v datasetu.	*+2b*
Konstrukce Minimum Area Enclosing box některou z metod (hlavní směry budov).	*+5b*
Algoritmus pro automatické generování konvexních/nekonvexních množin bodů různých tvarů (kruh, elipsa, čtverec, star-shaped, popř. další).	*+4b*
Max celkem:	**36b**

Obrázek 1.1: Zadání cičení č. 2 [2]

1.1 Údaje o bonusových úlohách

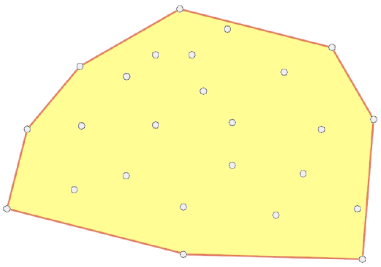
Cílem úlohy je představit grafickou aplikaci na konstrukci konvexních obálek nad různými množinami vygenorovaných bodů. Bonusovou úlohou je myšléná širší funkcionalita aplikace. Do aplikace byly zakomponovány následující prvky:

- *Konstrukce striktně konvexních obálek pro všechny uvedené algoritmy,*
- *Ošetření singulárního případu u Jarvis Scan: existence kolineárních bodů v datasetu,*
- *Algoritmus pro automatické generování konvexních/nekonvexních množin bodů různých tvarů (náhodné, pravidelná mřížka, kruh, elipsa, čtverec). [2]*

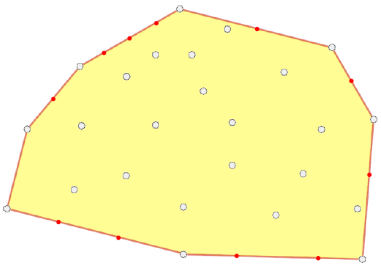
Kapitola 2

Popis problému

Mějme množinu bodů P v rovině. Chceme utvořit konvexní geometrický útvar, který má nejmenší obsah a body množiny P se nacházejí na jeho hraně nebo uvnitř. Takový útvar se nazývá konvexní obálkou viz. *Obrázek 2.1*. Dále se také pracuje s termínem striktně konvexní obálka viz. *Obrázek 2.2*. K určení konvexní obálky jsou použity algoritmy *Jarvis Scan*, *Quick Hull* a *Sweep Line*. Vstupní množina bodů je generována náhodně, ve zvoleném vzoru. Cílem je určit časovou náročnost algoritmů pro jednotlivé množiny. Principy jednotlivých algoritmů jsou popsané v následující kapitole 3.



Obrázek 2.1: Konvexní obálka [2]



Obrázek 2.2: Striktně konvexní obálka - vyloučí přebytečné body na obálce (červené body), ponechá pouze vrcholy [2], upravené autorem.

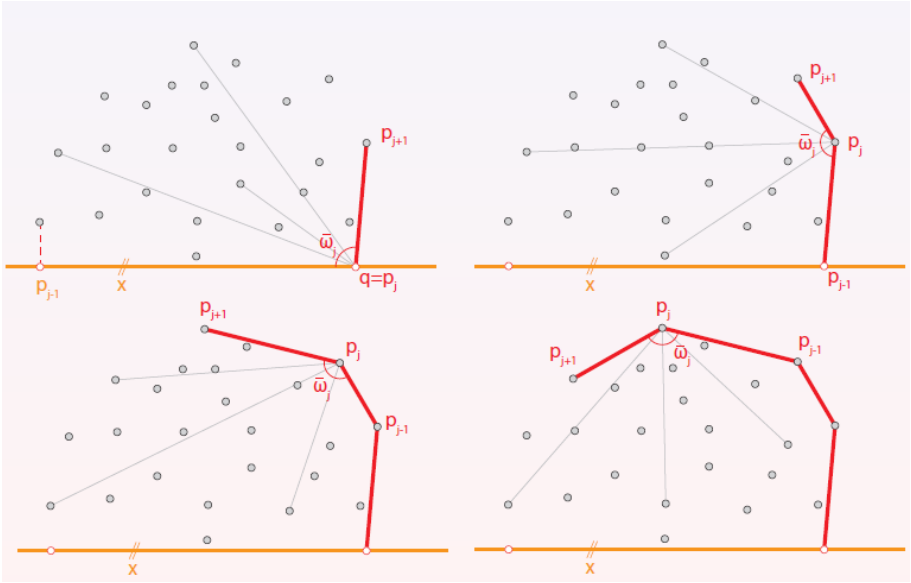
Kapitola 3

Popis použitých algoritmů

Pro vytvoření konvexní obálky pro množinu bodů byly použity algoritmy *Jarvis Scan*, *Quick Hull* a *Sweep Line*. Úlohu lze řešit také pomocí algoritmu Grahm Scam, jehož řešení v programu není použito.

3.1 Jarvis Scan

Metoda Jarvis Scan hledá bod P_{j+1}, pro který platí $\angle(P_{j-1}, P_j, P_{j+1})$ je maximální. Když takový bod nalezne, přidá jej do obálky a pokračuje v hledání následujícího bodu v obálce stejným způsobem, kde $P_j \equiv P_{j-1}$ a $P_{j+1} \equiv P_j$. Tento proces se opakuje, dokud nenajde bod, který byl do obálky přidán jako první.

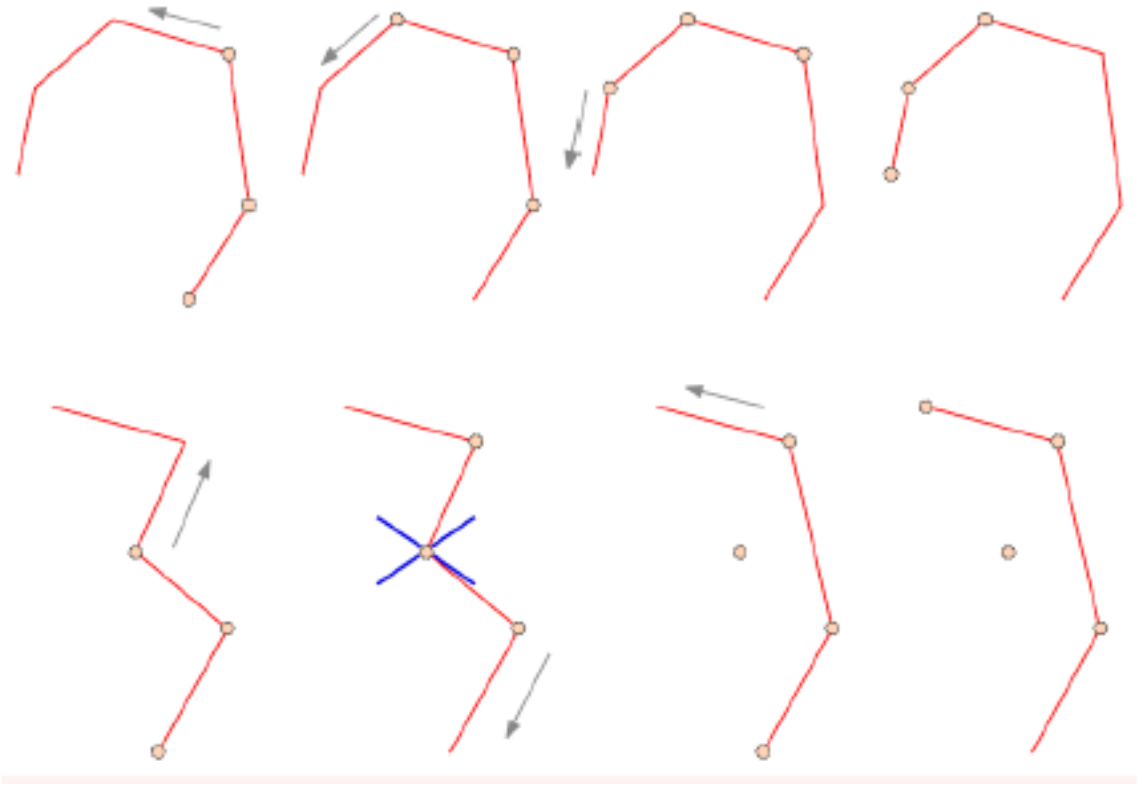


Obrázek 3.1: Princip Jarvis Scan Algorithm [1]

3.1.1 Problematické situace u Jarvis Scan Algorithm

V případě, že by agloritmus našel více variant následujícího bodu, tj. $\angle_i(P_{j-1}, P_j, P_{j+1}) \cong \angle_{i+1}(P_{j-1}, P_j, P_{j+1})$, nastává situace, kdy jsou dva po sobě jdoucí body kolineární. Tuto situaci je potřeba řešit tak, že se do konvexní obálky nejprve přidá bod který je nejblíže svému předchůdci - aktuálně poslednímu bodu obálky.

3.1.2 Implementace Jarvis Scan Algorithm

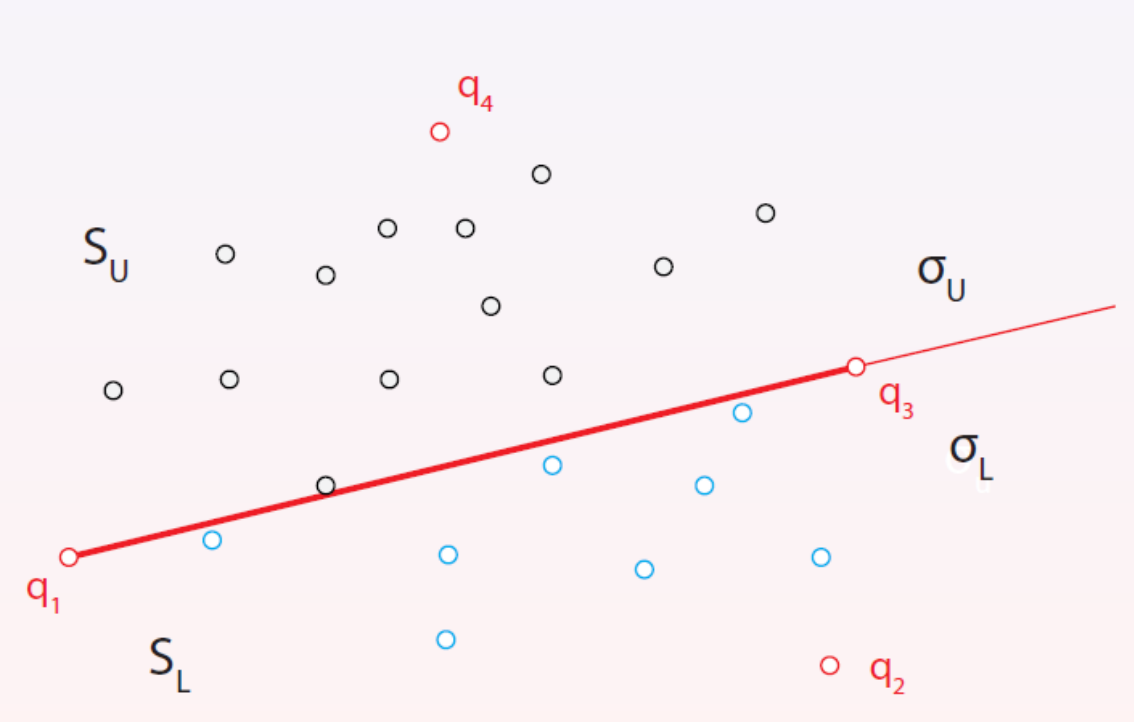


Obrázek 3.2: Implementace kritéria levotočivosti [1]

1. Nalezení pivota Q: $Q = min_{\forall p_i \in P}(Y_i)$
2. Přidej: $Q -> H$
3. Iniacializuj: $P_{j-1} \in X, P_j = Q, P_{j+1} = P_{j-1}$
4. Opakuj, dokud $P_{j+1} \neq Q$:
5. Nalezni $P_{j+1} = argmax_{\forall p_i \in P} \angle(P_{j-1}, P_j, P_{j+1})$
6. Přidej: $P_{j+1} -> H$
7. $P_j = P_{j-1}$; $P_{j+1} = P_j$

3.2 Quick Hull

Pro tuto metodu je zapotřebí nalézt dva body, které mají minimální P_1 resp. maximální P_2 $X-$ovou nebo $Y-$ovou souřadnici. Tyto body budou součástí konvexní obálky. Nejprve je však spojíme úsečkou a rozdělíme ostatní body podle toho, zda se nacházejí vlevo nebo vpravo od úsečky - $(P_i \in \Sigma_l || P_i \in \Sigma_p)$. Když budeme konvexní obálku tvořit s contra clock wise (CCW) orientací, přidáme do konvexní obálky bod P_1. Dále budeme hledat nejvzdálenější bodP_3 vpravo od orientované úsečky $|P_1P_2|$. Takový bod bude patřit do konvexní obálky. Než jej tam přidáme, tak musíme stejným způsobem zjistit, zda se mezi bodemP_1 aP_3 nenachází jiný bod. Pokud ne, tak bod P_3 přidáme do konvexní obálky a hledáme další bod, který je vpravo od úsečky $|P_3P_2|$. Když najdeme všechny body v této polorovině, tak hledáme body ve druhé polorovině stejným způsobem, ale přehodíme orientaci úsečky $|P_1P_2|$ na $|P_2P_1|$.



Obrázek 3.3: Princip Quick Hull Algorithm [1]

3.2.1 Implementace metody

Quick Hull algoritmus pracuje na lokální a globální úrovni. Postupně se na globální úrovni zpracuje horní a dolní část obálky rekurzivním voláním lokální procedury.

Globální procedura:

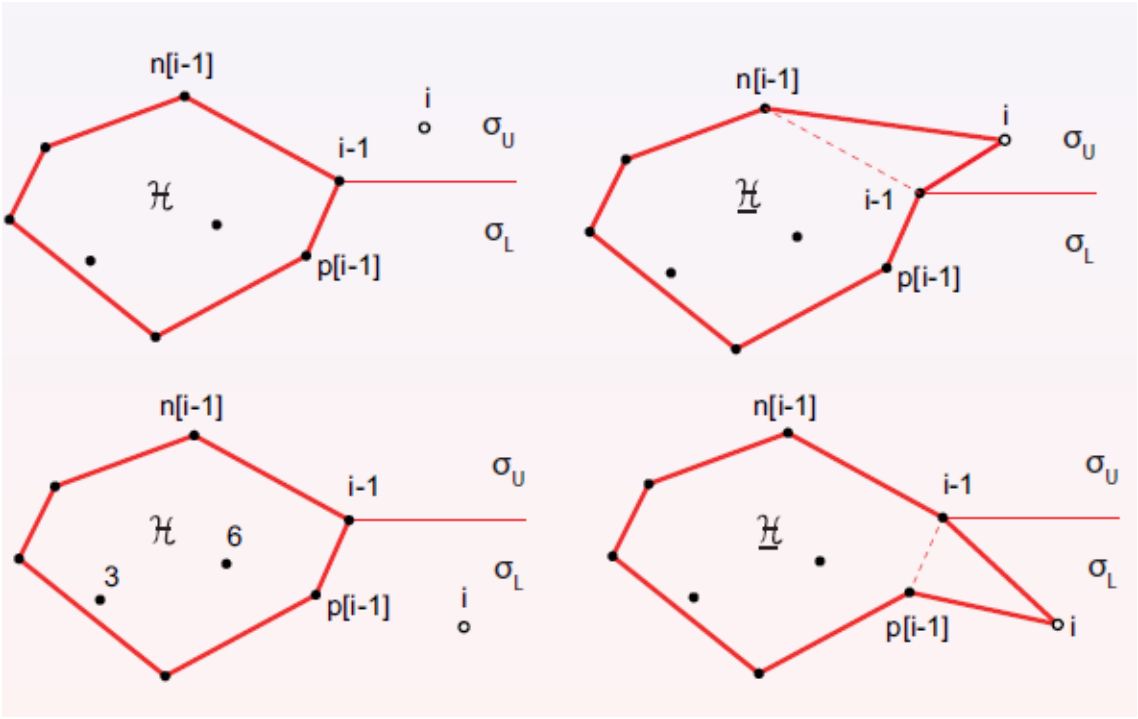
1. Inicializace: $H = 0, \Sigma_U = 0, \Sigma_L = 0$
2. Nalezení pivotů q: $q_1 = min_{\forall P_i \in \Sigma}(x_i)$, $q_3 = max_{\forall P_i \in \Sigma}(x_i)$
3. $\Sigma_U < -q_1$, $\Sigma_U < -q_3$
4. $\Sigma_L < -q_1$, $\Sigma_L < -q_3$
5. Pro $\forall P_i \in \Sigma$
6. Pokud: $(P_i \in \sigma_l(q_1, q_3))\Sigma_U < -P_i$
7. Jinak: $\Sigma_L < -P_i$
8. Přidej: $H < -q_3$
9. Vrchní množina: *Quick Hull* $(1, 0, \Sigma_U, H)$
10. Přidej: $H < -q_1$
11. Spodní množina: *Quick Hull* $(0, 1, \Sigma_U, H)$

Lokální procedura - *Quick Hull*:

1. Najdi bod $\bar{p} = argmax_{\forall P_i \in \Sigma}\ ||P_i - (P_s, P_e)||$, $\bar{p} \in \sigma_r(P_s, P_e)$
2. Pokud: $\bar{p} \neq 0 -> P_i \in \sigma_r$
3. *Quick Hull* $(s, \bar{i}, \Sigma, H)$ - Vrchní množina
4. $H < -\bar{p}$
5. *Quick Hull* $(\bar{i}, e, \Sigma, H)$ - Spodní množina

3.3 Sweep Line

Metoda Sweep Line rozděluje množinu bodů na zpracovanou a nezpracovanou. Množiny jsou rozděleny přímkou, většinou rovnoběžnou s některou souřadnicovou osou. Body je tedy potřeba seřadit podle této osy. Vyhodnocování probíhá postupně pro každý bod.



Obrázek 3.4: Princip Quick Hull Algorithm [1]

3.3.1 Problematické situace u Sweep Line Algorithm

Problémy u této metody nastanou v případě výskytu duplicitních bodů, proto je dobré ještě před započetím výpočtu tyto body odstranit.

3.3.2 Implementace metody

U metody Sweep line se velký důraz klade na správné indexování, proto je důležité chápat všech šest možných pozic v množině bodů.

Indexy algoritmu:

- p[i] předchůdce i-tého vrcholu, (i-1)-ty vrchol.
- n[i] následník i-tého vrcholu, (i+1)-ty vrchol.

- p[p[i]] předchůdce předchůdce, (i-2)-ty vrchol.
- n[n[i]] následník následníka, (i+2)-ty vrchol.
- n[p[i]] následník předchůdce, i-ty vrchol.
- p[n[i]] předchůdce následníka, i-ty vrchol.

Sweep Line Algorithm:

1. Seřaď: $P_\Sigma = sort(P_i)$ *podle* X
2. Pokud: $(P_3 \in \sigma_L(P_1; P_2))$
3. *n[1] = 2, n[2] = 3, n[3] = 1*
4. *p[1] = 3, p[2] = 1, p[3] = 2*
5. Jinak:
6. *n[1] = 3, n[3] = 2; n[2] = 1*
7. *p[1] = 2, p[3] = 1; p[2] = 3*
8. Pro: $P_i \in P_\Sigma$,$i > 3$
9. Pokud: (Y_i>Y_{i-1})
10. *p[i] = i-1, n[i] = n[i-1]*
11. Jinak:
12. *n[i] = i-1, p[i] = p[i-1]*
13. *n[p[i]] = i, p[n[i]] = i*
14. Dokud: (n[n[i]]) $\in \Sigma_R$ (i; n[i])
15. *p[n[n[i]]] = i, n[i] = n[n[i]]*
16. Dokud: (p[p[i]]) $\in \Sigma_L$ (i, p[i])
17. *n[p[p[i]]] = i, p[i] = p[p[i]]*

Kapitola 4

Vstup a výstup aplikace

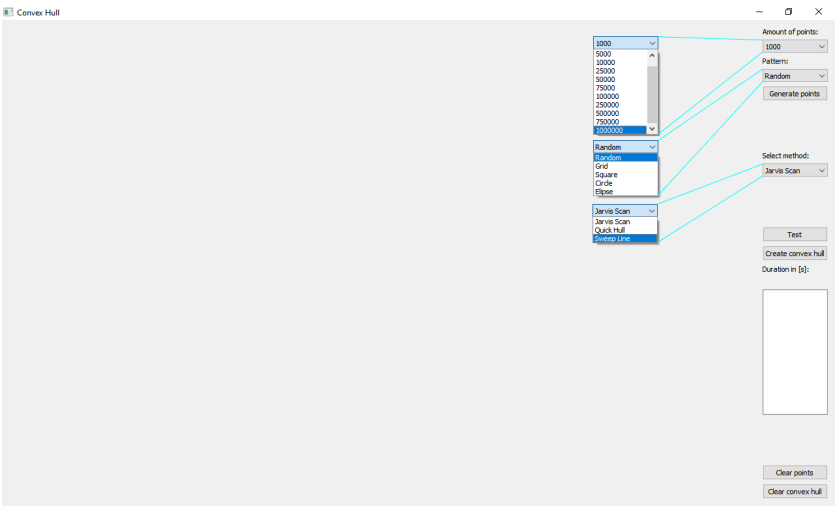
4.1 Vstup

Vstupními daty pro tuto úlohu je množina bodů, kterou uživatel může vytvořit kliknutím myši do grafické části aplikace. Druhá možnost jak vytvořit body je automatikou generací, kdy uživatel zvolí počet bodů, který chce vytvořit z nabídky možností od 1 000 až po 1 000 000 bodů a dále zvolí, zda mají být body vygenerovány náhodně, v pravidelném rastru, na kružnici, na elipse nebo ve tvaru čtverce.

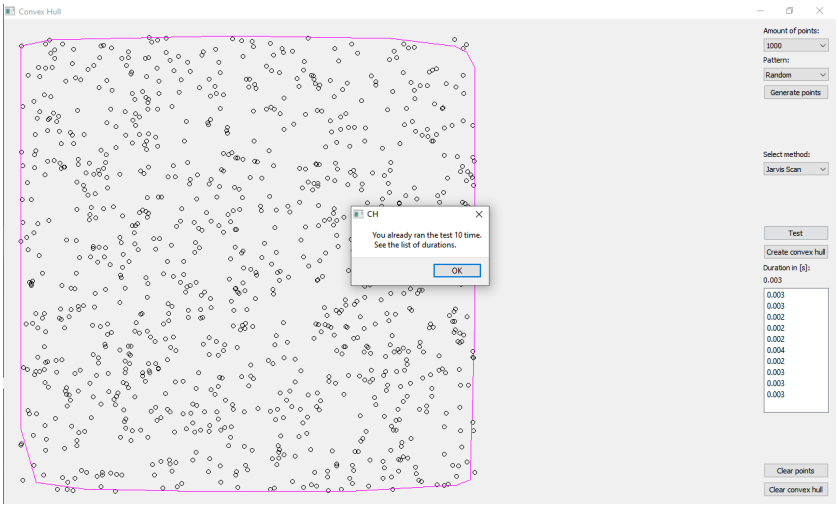
4.2 Výstup

Výstupem úlohy je grafická aplikace, která ze zadaných nebo vygenerovaných bodů vytvoří konvexní obálku. Konvexní obálku lze vytvořit pomocí algoritmů Jarvis Scam, Quick Hull nebo Sweep Line. Po provedení se vytvoří polygon konvexní obálky. Výstupem je také časový údaj doby, kterou algoritmus potřeboval pro výpočet obálky. Z časových údajů jsou vytvořené grafy a tabulky jako další výstup úlohy.

4.2.1 Prostředí programu

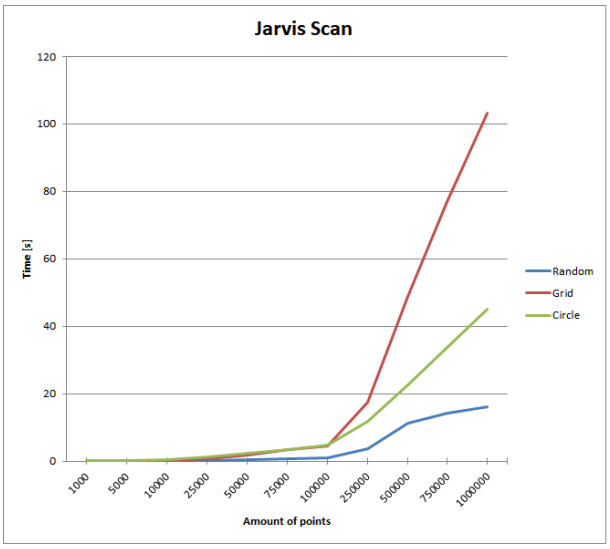


Obrázek 4.1: Prostředí po spuštění aplikcace - možné volby



Obrázek 4.2: Hláška ukočení testu trvání generování a vykreslení algoritmu

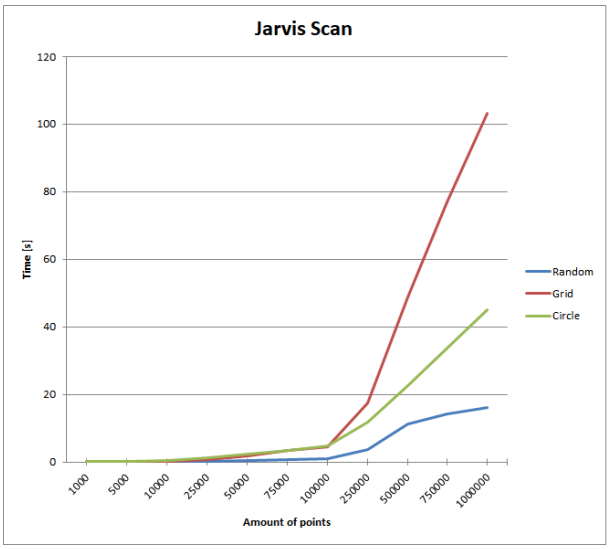
4.2.2 Výpočetní čas



Obrázek 4.3: Jarvis Scan - vcšechny množiny

Jarvis Scan			
Počet bodů	Random[s]	Grid[s]	Circle[s]
1000	0.003	0.005	0.027
5000	0.018	0.056	0.228
10000	0.041	0.177	0.449
25000	0.122	0.611	1.177
50000	0.344	1.711	2.321
75000	0.796	3.289	3.519
100000	1.005	4.378	4.686
250000	3.640	17.540	11.671
500000	11.327	48.921	22.719
750000	14.176	76.909	33.571
1000000	16.136	103.310	44.987

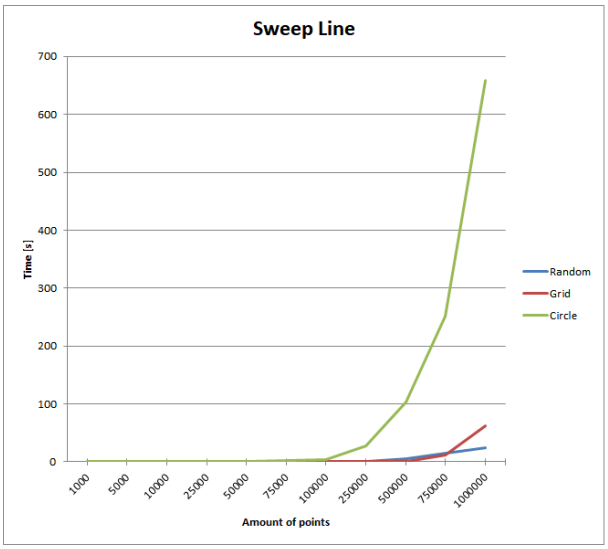
Obrázek 4.4: Jarvis Scan - vcšechny množiny, hodnoty



Obrázek 4.5: Quick Hull - vcšechny množiny

Quick Hull			
Počet bodů	Random[s]	Grid[s]	Circle[s]
1000	0.000	0.000	0.001
5000	0.001	0.001	0.006
10000	0.001	0.001	0.009
25000	0.003	0.002	0.027
50000	0.006	0.003	0.048
75000	0.007	0.005	0.073
100000	0.010	0.007	0.087
250000	0.018	0.017	0.227
500000	0.028	0.037	0.433
750000	0.032	0.072	0.672
1000000	0.036	0.085	0.900

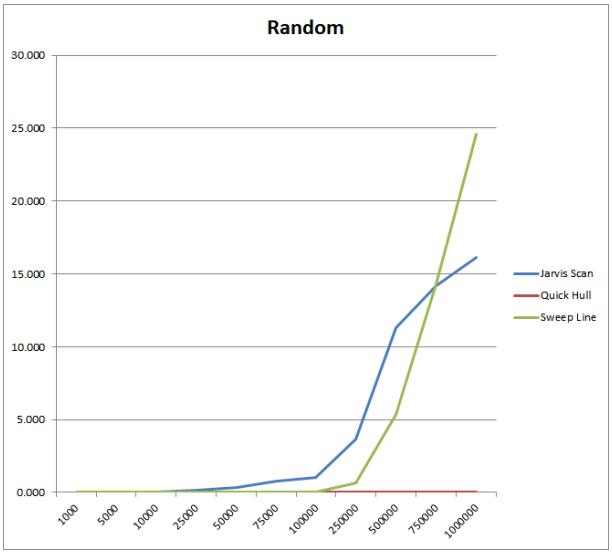
Obrázek 4.6: Quick Hull - vcšechny množiny, hodnoty



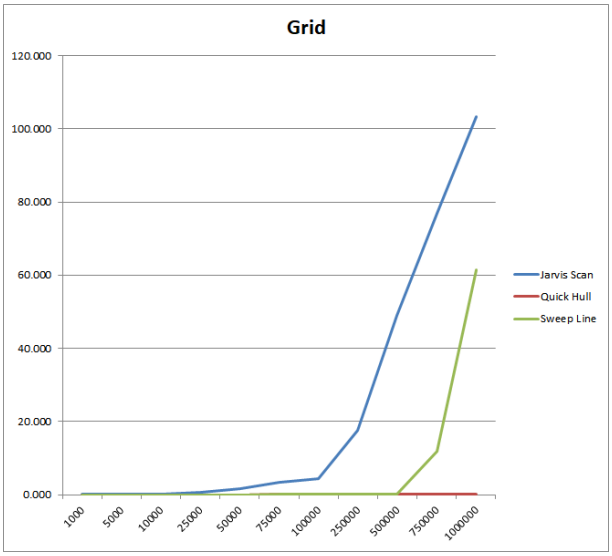
Obrázek 4.7: Sweep Line - vcšechny množiny

Sweep Line			
Počet bodů	Random[s]	Grid[s]	Circle[s]
1000	0.000	0.000	0.000
5000	0.000	0.000	0.003
10000	0.001	0.000	0.023
25000	0.001	0.001	0.202
50000	0.002	0.002	0.907
75000	0.004	0.004	2.068
100000	0.010	0.005	3.748
250000	0.650	0.015	27.308
500000	5.381	0.044	103.397
750000	14.337	11.863	251.260
1000000	24.595	61.535	658.957

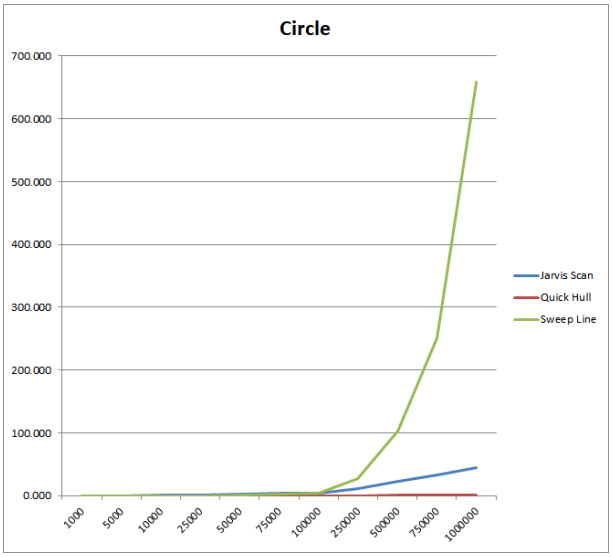
Obrázek 4.8: Sweep Line - vcšechny množiny, hodnoty



Obrázek 4.9: Všechny algoritmy - množina náhodných bodů



Obrázek 4.10: Všechny algoritmy - množina grid bodů



Obrázek 4.11: Všechny algoritmy - množina kruhových bodů

Kapitola 5

Dokumentace tříd a jejich metod

5.1 Algorithms

- *double getPointLineDistance(QPoint &q, QPoint &p1, QPoint &p2)*
  Tato funkce počítá vzdálenost bodu q od přímky. Přímka je zadaná dvěma body. Na vstupu funkce jsou tři body a výstupem je vzdálenost typu double.
- *int getPointLinePosition(QPoint &q,QPoint &p1,QPoint &p2)*
  Tato funkce určuje pozici bodu vůči linii. Na vstupu funkce je určovaný bod a dva body přímky "a"a "b". Ze dvou bodů přímky můžeme určit determinant, jehož hodnotu porovnáváme se zvolenou minimální hodnotou "eps". Výstupem funkce je celé číslo, které nabývá hodnot 1(pod leží v levé polorovině), 0(bod leží v pravé polorovině) a -1(bod leží na hraně).
- *double getAngle2Vectors(QPoint &p1,QPoint &p2,QPoint &p3,QPoint &p4)*
  Tato funkce počítá úhel mezi dvěma hranami. Na vstupu jsou 4 body, které určují dvě přímky. Výstupem je velikost úhlu ve stupních typu double.
- *double getLength2Points(QPoint q, QPoint p)*
  Tato funkce počítá vzdálenost mezi dvěma body. Na vstupu funkce jsou dva body a výstupem je vzdálenost typu double.
- *QPolygon jarvisScan(std::vector<QPoint> &points)*
  Funkce, která metodou Jarvis Scan zjišťuje konvexní obálku množiny bodů. Na vstupu je množina všech bodů. Výstupem je množina bodů tvořící polygon.
- *QPolygon qHull(std::vector<QPoint> &points)*
  Funkce, která metodou Quick Hull zjišťuje konvexní obálku množiny bodů. Na vstupu je množina všech bodů. Výstupem je množina bodů tvořící polygon.
- *void qh(int s, int e, std::vector<QPoint> &points, QPolygon &ch)*
  Pomocná funkce pro určení konvexní obálky metodou Quick Hull. Jedná se o rekurzivní funkci, hledající nejvzdálenější bod vpravo od zadané úsečky. Na vstupu je počáteční bod úsečky, koncový bod úsečky, množina bodů, a body v konvexní obálce. Funkce nemá žádnou návratovou hodnotu, pouze ukládá nalezený bod do množiny bodů konvexní obálky.

- *QPolygon sweepLine(std::vector<QPoint> &points)*
  Funkce, která metodou Sweep Line zjišťuje konvexní obálku množiny bodů. Na vstupu je množina všech bodů. Výstupem je množina bodů tvořící polygon.

- *QPolygon strictlyCH(QPolygon ch)*
  Funkce, která tvoří striktně konvexní obálku. Vstupem je množina bodů tvořící konvexní obálku. Výstupem je tatáž množina bez bodů, které leží na úsečce mezi předchozím a následujícím bodem obálky.

5.2 Draw

- *void mousePressEvent(QMouseEvent *event)*
  Metoda, která slouží ke vkládání bodů po kliknutí do grafického okna.

- *void paintEvent*
  Metoda, která vykresluje vygenerované body a konvexní obálku.

- *void clearCH()*
  Slouží k vyčištění grafického okna od konvexní obálky.

- *void clearPoints()*
  Slouží k vyčištění grafického okna od všech bodů.

- *void setCH(QPolygon &hull)*
  Metoda, která převádí konvexní obálku do grafického okna.

- *std::vector<QPoint>generatePointsRandom(int n_points)*
  Metoda, která generuje množinu náhodných bodů. Na vstupu je počet bodů, který má množina obsahovat. Výstupem je vektor bodů.

- *std::vector<QPoint>generatePointsGrid(int n_points)*
  Metoda, která generuje množinu bodů v pravidelném gridu. Na vstupu je počet bodů, který má množina obsahovat. Výstupem je vektor bodů.

- *std::vector<QPoint>generatePointsSquare(int n_points)*
  Metoda, která generuje množinu bodů ve tvaru čtverce. Na vstupu je počet bodů, který má množina obsahovat. Výstupem je vektor bodů.

- *std::vector<QPoint>generatePointsCircle(int n_points)*
  Metoda, která generuje množinu bodů ve tvaru kruhu. Na vstupu je počet bodů, který má množina obsahovat. Výstupem je vektor bodů.

- *std::vector<QPoint>generatePointsEllipse(int n_points)*
  Metoda, která generuje množinu bodů ve tvaru elipsy. Na vstupu je počet bodů, který má množina obsahovat. Výstupem je vektor bodů.

- *std::vector<QPoint>generatePointsStar(int n_points)*
  Metoda, která generuje množinu bodů ve tvaru hvězdy. Na vstupu je počet bodů, který má množina obsahovat. Výstupem je vektor bodů.

- *void setPoints(std::vector<QPoint> gen_points)*
  Metoda, která převádí body z grafického okna.

- *double saveTime(std::vector<double> times)*
  Metoda, která ukládá čas výpočtu algoritmu.

5.3 Widget

- *void on_pushButton_clicked*
  Spustí výpočet algoritmu pro určení konvexní obálky. V aplikaci se jedná o tlačítko Create convex hull.

- *void on_pushButton_2_clicked*
  Smaže množinu bodů. V aplikaci se jedná o tlačítko Clear points.

- *void on_pushButton_3_clicked*
  Smaže konvexní obálku. V aplikaci se jedná o tlačítko Clear convex hull.

- *void on_generate_clicked*
  Vygeneruje množinu bodů. V aplikaci se jedná o tlačítko Generate points.

5.4 sortbyy

Třída sortbyy slouží k setřídění bodů podle Y-ové souřadnice.

- *bool operator() (QPoint &p1, QPoint &p2)*
  Z dvojice bodů vrátí ten s větší Y-ovou souřadnicí.

Kapitola 6

Závěr

Ve vytvořené aplikaci lze vygenerovat množinu bodů čtyřmi různými způsoby a vytvořit jejich konvexní obálku třemi různými metodami. Aplikace pak zvolenou kombinaci znázorní v grafickém prostředí. Program řeší i případy, kdy se na jedné hraně nalézá více bodů. Efektivnost dané kombinace je možno otestovat z hlediska výpočetního času.
U algoritmu Sweep Line převzatého z cvičení byla objevena nedokonalost v generování obálky ve směru od dolní hrany k horní. Problém se napodalo vyřešit a do budoucna bude nutné jej opravit.

6.1 Neřešené problémy a náměty

- Z časové tísně nebyla v úloze vyřešena konstrukce konvexní obálky metodou Graham Scan a konstrukce Minimum Area Enclosing box pro žádnou z metod.
- Data jsou předem velikostně definována, jelikož program má předdefinovanou velikost vykreslovacího okna. V ideálním případě by bylo dobré, kdyby se načtená data sama přízpůsobila velikosti okna.
- Praktické by též bylo rozšířit aplikaci o funkci vykreslující grafy časové náročnosti přímo v grafickém prostředí.

Literatura

[1] T. Bayer. Konvexní obálka množiny bodů, přednáška č. 4 předmětu Algoritmy v digitální kartografii, 2016. `https://web.natur.cuni.cz`.

[2] T. Bayer. Konvexní obálky a jejich konstrukce, cvičení č. 2 předmětu Algoritmy v digitální kartografii, 2016. `https://web.natur.cuni.cz`.

Příloha A

Zdrojový kód aplikace

Zdrojový kód aplikace je dostupný z veřejného profilu *petrposkocil* na github.com
Jméno repozitáře:*ADK_poskocil_faber*
Podsložka: *u2_ch*

https://github.com/petrposkocil/ADK_poskocil_faber/U2/U2_ch

Obsažené soubory:
CH.pro //Qt creator project file
algorithms.h //Header file
draw.h //Header file
sortbyx.h //Header file
sortbyy.h //Header file
widgets.h //Header file
main.cpp //Source code file
algorithms.cpp //Source code file
draw.cpp //Source code file
sortbyx.cpp //Source code file
sortbyy.cpp //Source code file
widgets.cpp //Source code file
widget.ui //User interface file

Příloha B

Aplikace - binární soubor

Aplikace je dostupná z veřejného profilu *petrposkocil* na github.com
Jméno repozitáře: *ADK_ poskocil_ faber*
Soubor: *CH.exe*

https://github.com/petrposkocil/ADK_poskocil_faber/U2/CH.exe

Příloha C

Výstupní data

Výstupní data jsou ve fromě grafického znázornění kovexní obálky nad množinou bodů.